# Introducing micro:bian

Mike Spivey

Hilary Term 2022


Department of
COMPUTER
SCIENCE

# In this part

- Concurrent processes and messages between them as a way of structuring complex systems that respond to events (L12).
- Managing I/O devices with driver processes that receive interrupts as messages (L13).
- Implementing multiple processes (L14).
- Messages and scheduling (L15).
- Chasing down a bug (L16).

# Why concurrency?

- Genuinely parallel machines
- Sharing one machine between several tasks
- Decomposing one task clearly
- Responding to several sources of events

# In this lecture

- *Processes*: embedded programs are conveniently structured as a set of independent processes.
- *Messages*: processes can cooperate by exchanging messages in a way that synchronises their behaviour.
- *Shared variables* are best avoided by using messages instead.

# Hearts again

```
static int row = 0;

void advance(void) {
    row++;
    if (row == 3) row = 0;
    GPIO_OUT = heart[row];
}
```

- Efficient but inflexible.
- Can’t pause inside subroutines or control structures.

# But also primes

Use interrupts to overlap printing with the search, but ...

- When the serial buffer is full, wastes time waiting in a loop.
- Disables interrupts to protect the buffer from concurrent modification – hard to get right.

We’re ready for to use an operating system: enter micro:bian!

# Better: a process

```
static void heart_task(int arg) {
    while (1) {
        show(heart, 70);
        show(small, 10);
        show(heart, 10);
        show(small, 10);
    }
}
```

# Better: a process

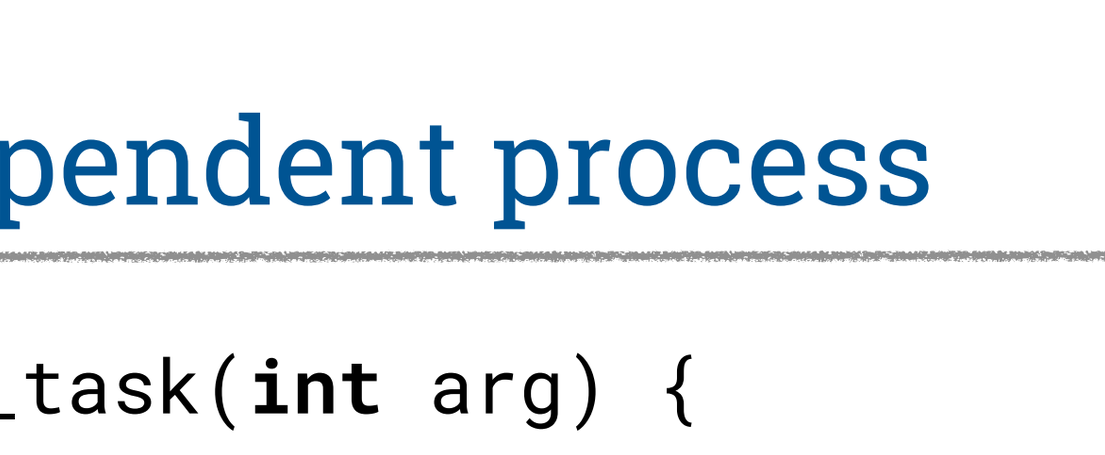
```
static void heart_task(int arg) {
    while (1) {
        show(heart, 70);
        show(small, 10);
        show(heart, 10);
    }
}
```

```
static void show(int img[], int n) {
    while (n-- > 0) {
        for (int p = 0; p < 3; p++) {
            GPIO_OUT = img[p];
            timer_delay(5);
        }
    }
}
```

# Another, independent process

```
static void prime_task(int arg) {
    int p = 2, n = 0;

    while (1) {
        if (prime(p)) {
            n++;
            printf("prime(%d) = %d\n", n, p);
        }
        p++;
    }
}
```



# Setting the ball rolling

```
void init(void) {
    SERIAL = start("Serial", serial_task, 0, STACK);
    TIMER = start("Timer", timer_task, 0, STACK);
    HEART = start("Heart", heart_task, 0, STACK);
    PRIME = start("Prime", prime_task, 0, STACK);
}
```

- a fixed collection of processes created before concurrent execution begins.
- our two processes, plus *device drivers* for the timer (`timer_delay`) and serial port (`serial_putc`); plus an idle task.

# Processes

Each a ‘main program’ in its own right

- It can call subroutines.
- It can pause (or be interrupted) at any point to give others a go.

Implementation

- Processes are interleaved.
- Each has its own stack.

micro:bian supports a fixed set of processes.

# Other operating systems

- Processes with communication
- Memory management
- Drivers for I/O devices
- File system
- Networking

micro:bian supports processes and messages, and whatever device drivers we write.

No utility programs, shared libraries, GUI, ... either.

# Sending messages

```
void prime_task(int arg) {
    int n = 2;
    message m;

    while (1) {
        if (prime(n)) {
            m.int1 = n;
            send(USEPRIME, PRIME, &m);
        }
        n++;
    }
}
```

# Receiving messages

```
void summary_task(int arg) {
    int count = 0, limit = arg; message m;

    while (1) {
        receive(PRIME, &m);
        while (m.int1 >= limit) {
            report(count, limit);
            limit += arg;
        }
        count++;
    }
}
```

# Rules for messages

Both sender and receiver have a message buffer (16 bytes).

- The sender assembles a message; then
- It is transferred from sender to receiver as an *atomic* action.
- No buffering, no queues of messages!

# Alternatives to messages

Message passing:

- no “shared variables” between processes.
- all communication by messages

Shared variables with semaphores:

- like the serial output buffer.
- more efficient, but hard to get right.

# Device drivers

Mike Spivey

Hilary Term 2022


Department of
COMPUTER
SCIENCE

# In this lecture

- *Interrupts* can be tamed by turning them into ‘messages from the hardware’.
- *Device drivers* look after hardware devices by serving requests one at a time in a loop.

*(See wiki and Lab 4 for all details – many are omitted here for clarity.)*

# Implementing serial output

```
void serial_putc(char ch) {
    message m;
    m.int1 = ch;
    send(SERIAL, PUTC, &m);
}
```

- request message sent to the `SERIAL` driver.
- the caller *waits* if the driver is not ready.

# Implementing the driver process

```
void serial_task(int arg) {
    static char txbuf[NBUF];
    int bufin, bufout, bufcount;
    message m; char ch;

    serial_setup();

    while (1) {
        receive(ANY, &m);
        switch (m.m_type) {
            ...
        }
    }
}
```

# Implementing the driver process

```
void serial_task(int arg) {
    static char txbuf[NBUF];
    int bufin, bufout, bufcount;
    message m; char ch;

    serial_setup();

    while (1) {
        receive(ANY, &m);
        switch (m.m_type) {
            ...
        }
    }
}
```

State is local to the driver

# Implementing the driver process

```
void serial_task(int arg) {
    static char txbuf[NBUF];
    int bufin, bufout, bufcount;
    message m; char ch;

    serial_setup();

    while (1) {
        receive(ANY, &m);
        switch (m.m_type) {
            ...
        }
    }
}
```

State is local to the driver

A server loop accepts requests

# Setting things up

```
void serial_setup(void) {
    ...

    connect(UART_IRQ);
    enable_irq(UART_IRQ);
    UART_INTENSET = BIT(UART_INT_TXDRDY);
}
```

# Handling PUTC messages

```
while (1) {
     receive(ANY, &m);
     switch (m.m_type) {
     case PUTC:
          ch = m.int1;
          txbuf[bufin] = ch; ...
          break;

          . . .
     }
}
```

- Buffer variables are local, so no other process can interfere.

# Handling interrupts

**Key insight:**

*an interrupt is a message from the hardware.*

```
receive(ANY, &m);
switch (m.m_type) {
case INTERRUPT:
 if (UART_TXDRDY) {
     txidle = 1;
     UART_TXDRDY = 0;
 }
clear_pending(UART_IRQ);
enable_irq(UART_IRQ);
 break;
```

# Responding to events

```
while (1) {
        receive(ANY, &m);
        switch (m.m_type) {
                    ...
        }

        if (txidle && bufcount > 0) {
                UART_TXD = txbuf[bufout]; ...
                txidle = 0;
        }
}
```

# When the buffer is full

Let's replace

```
receive(ANY, &m);
```

with

```
if (bufcount < NBUF)
    receive(ANY, &m);
else
    receive(INTERRUPT, &m);
```

When the buffer is full, we just stop accepting requests until it has emptied a bit.

# Omitted here …

Lab 4 has a more elaborate serial driver

- Supports both output and input with echoing and line editing.
- All UART initialisation details are filled in.
- There's an alternative interface `print_buf` that overcomes the one-message-per-character bottleneck.

# The standard interrupt handler

```
void default_handler(void) {
    int irq = active_irq();
    int task = os_handler[irq];
    disable_irq(irq);
    interrupt(task);
}
```

# Implementing Processes

Mike Spivey

Hilary Term 2022


Department of
COMPUTER
SCIENCE

# Concurrent processes

We want multiple processes, each with its own stack.  For simplicity,

- A fixed set of processes, created at the start.
- Each process has a fixed amount of stack space.

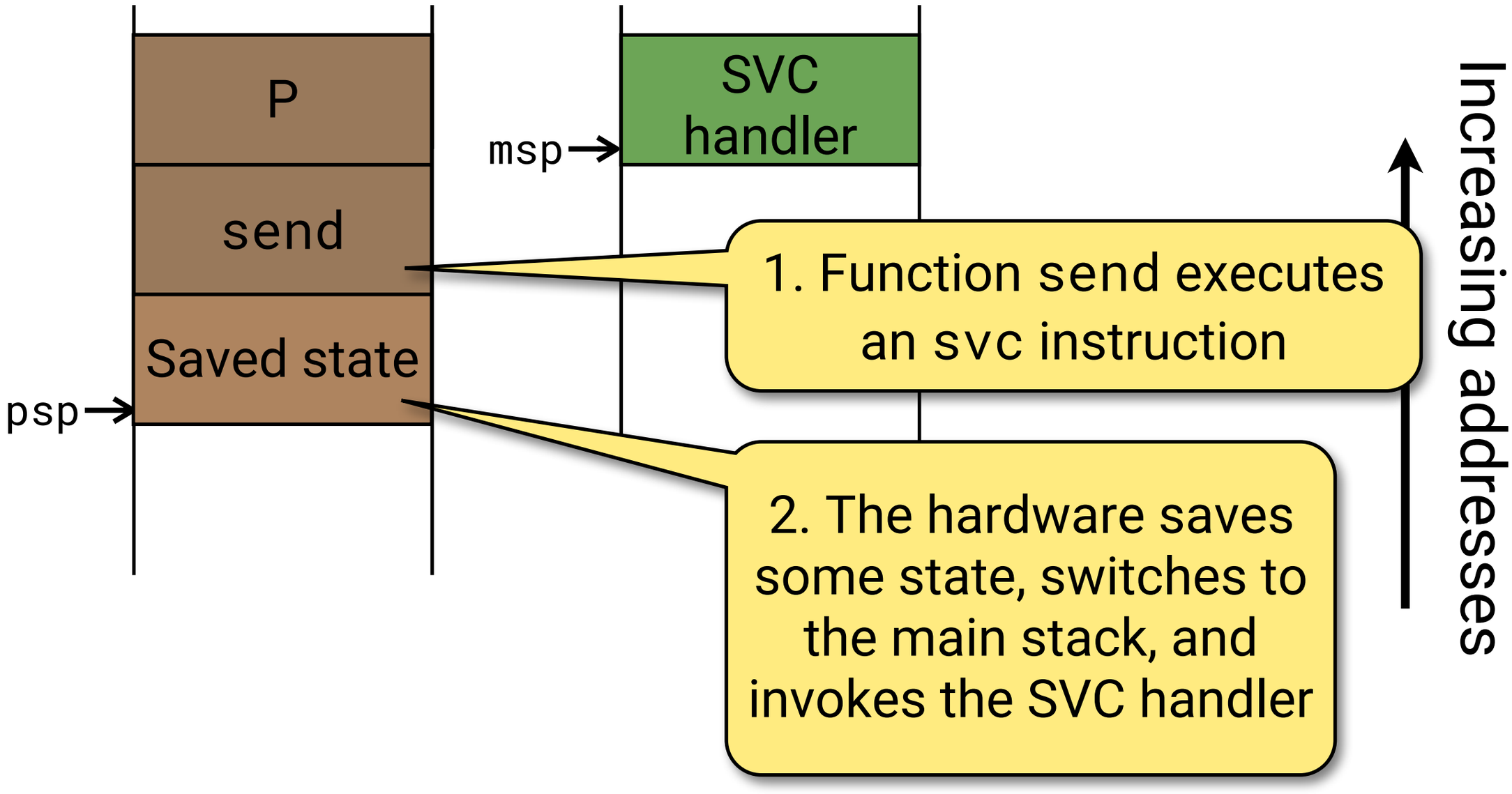
# Implementing processes

The plan:

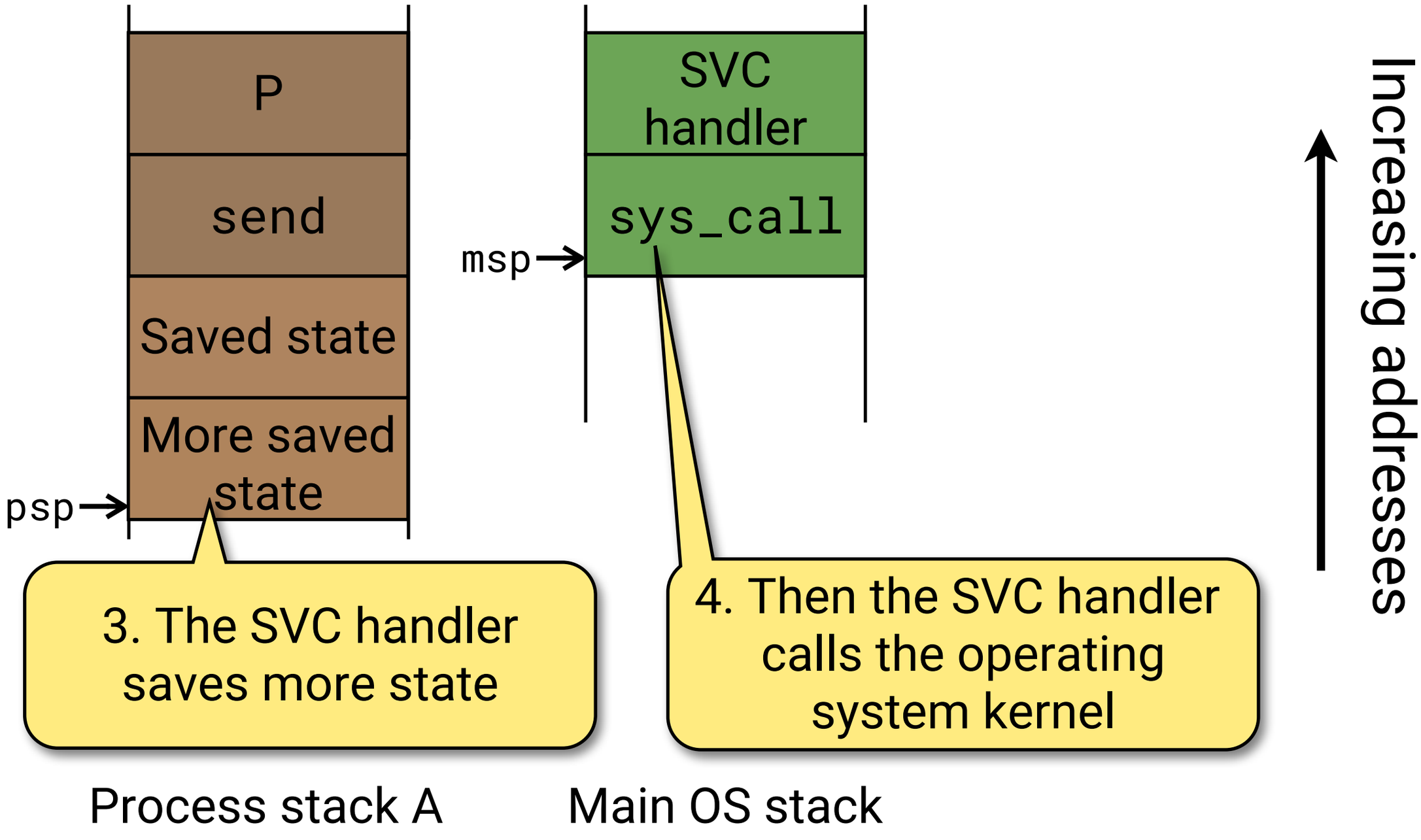
- Enter the OS via a *software interrupt* instruction `svc`, or by a normal interrupt
- Save the entire processor state on the stack
- After choosing a new process, restore its state to continue.

Made easier by having a *separate stack* for the operating system.

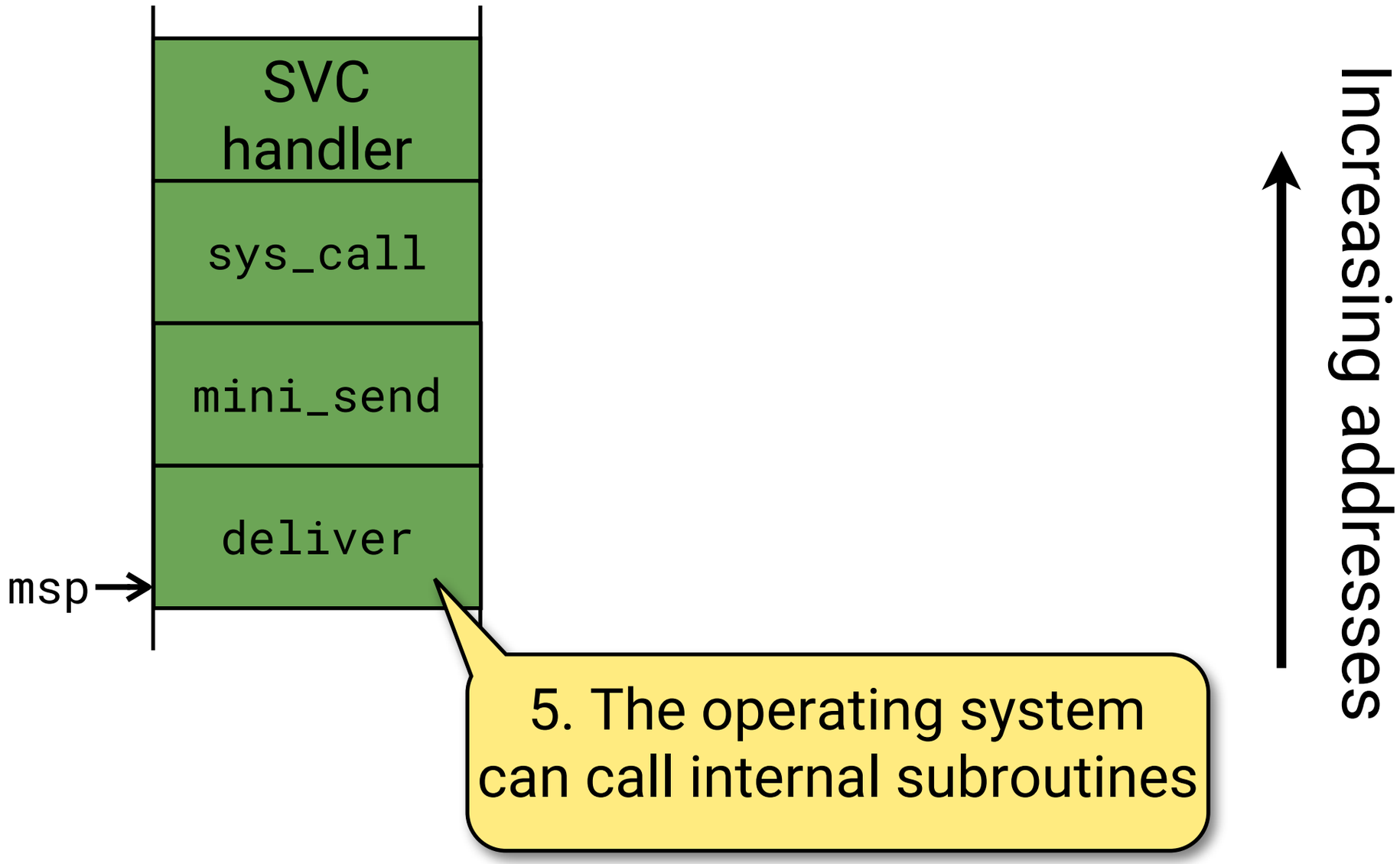
# Context switch – part 1



# Context switch – part 2

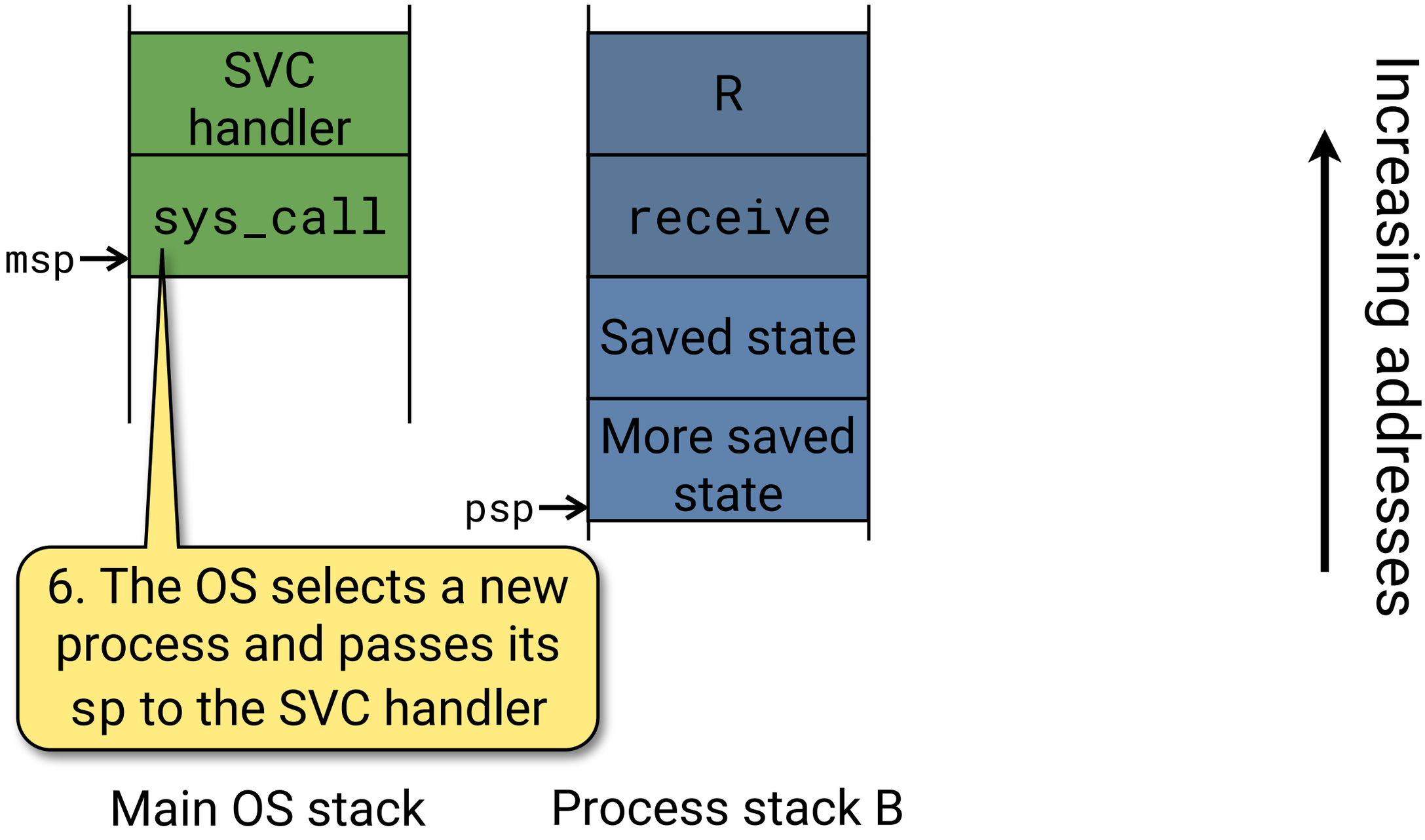


# Context switch – part 3

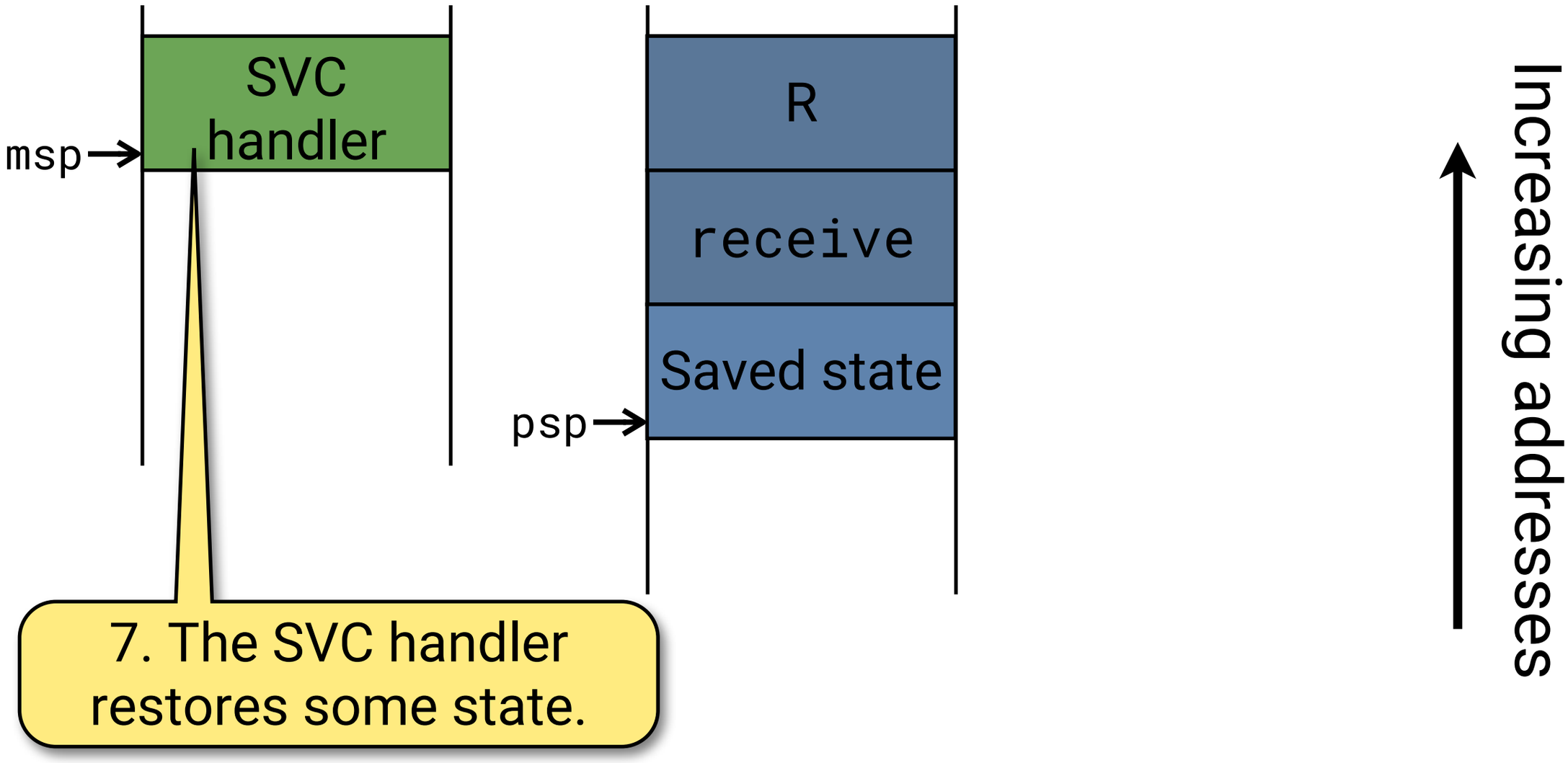



Main OS stack

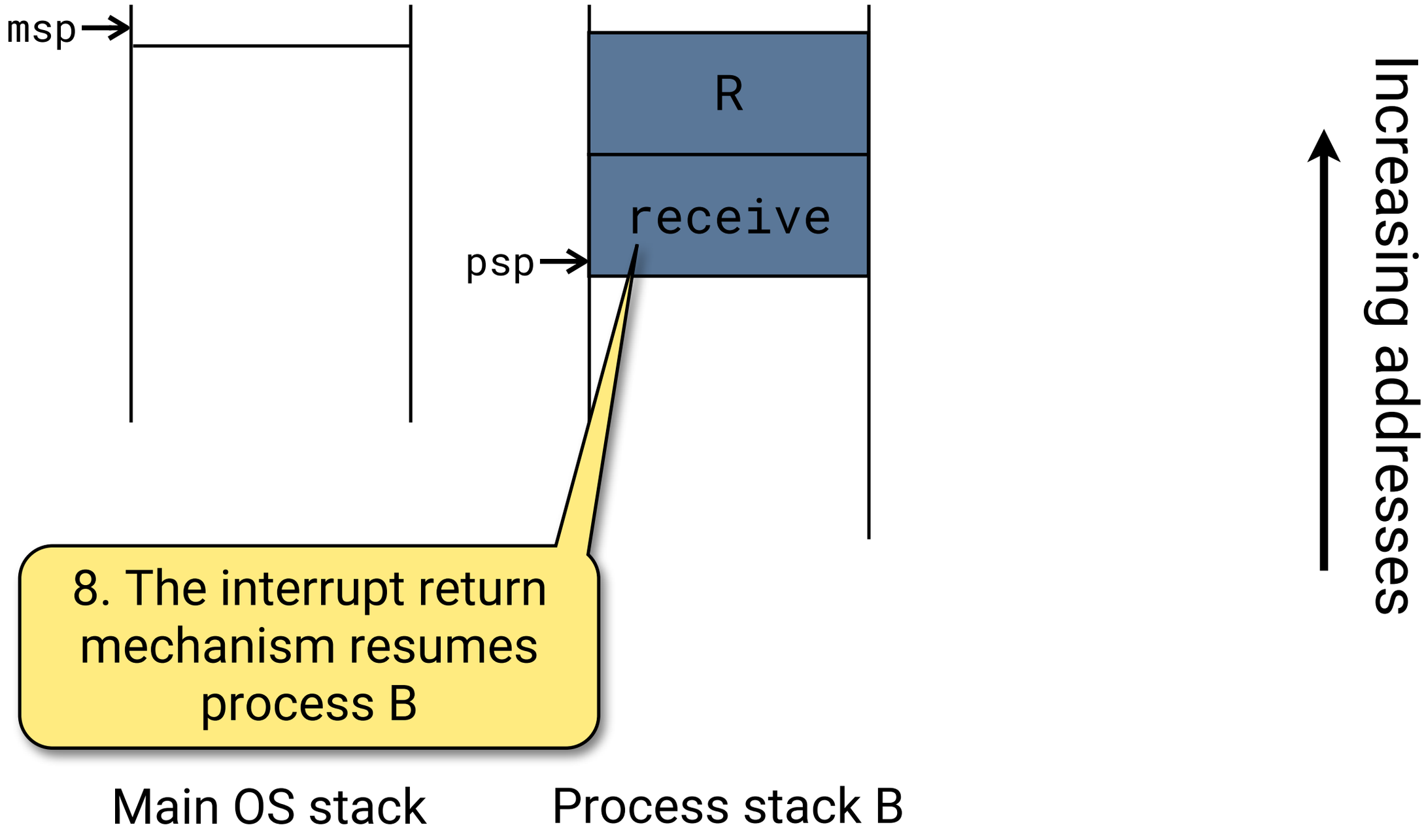
# Context switch – part 4



# Context switch – part 5



# Context switch – part 6



# System calls – client side

```
void NOINLINE yield(void) {
    syscall(SYS_YIELD);
}

void NOINLINE send(int dest, int type,
                   message *msg) {
    syscall(SYS_SEND);
}
```

Generates an `svc` instruction

OS will find arguments in registers `r0-r2`

# Implementing the SVC handler

```
svc_handler:
    isave               @ Complete saving of state
    @@ Argument in r0 is sp of old process
    bl system_call @ Perform system call
    @@ Result in r0 is sp of new process
    irestore            @ Restore saved state
```

(in `mpx-m0.s`)

# Saving the state

```
@@@ isave -- save context for system call
    .macro isave
    mrs r0, psp             @ Get thread stack pointer
    subs r0, #36
    movs r1, r0
    mov r3, lr              @ Preserve magic value
    stm r1!, {r3-r7}        @ Save low regs on thread stack
    mov r4, r8              @ Copy from high to low
    mov r5, r9
    mov r6, r10
    mov r7, r11
    stm r1!, {r4-r7}        @ Save high regs on thread stack
    .endm                   @ Return new thread sp
```

# System calls – OS side

```
unsigned *system_call(unsigned *psp) {
    short *pc = (short *) psp[PC_SAVE];
    int op = pc[-1] & 0xff;

    os_current->sp = psp;

    switch (op) {
    case SYS_YIELD:
        make_ready(os_current);
        choose_proc();
        break;
        ...
    }

    return os_current->sp;
}
```

# Completing the story

Two details remain:

- How to start a process.
- How to start the entire operating system.

# Starting a process

The first time a process runs, it is resumed just as if returning from a system call.

So we set up a fake exception frame that invokes the process body when resumed.

- `r0` = integer argument,
- `pc` = process body,
- `lr` = address of `exit` stub, in case body returns.

# Starting the system

After creating all the processes that make up the program, the main program becomes the idle process.

```
void __start(void) {
    /* Create idle process */
    ...
    /* Call the application's setup function */
    init();

    /* The main program morphs into the idle process. */
    os_current = idle_proc;
    set_stack(os_current->sp);
    idle_task();
}
```

# The idle process

Having an idle process saves us from ever having no process to run.

```
/* idle_task -- body of the idle process */
void idle_task(void) {
    /* Pick a genuine process to run */
    yield();

    /* When there's nothing to do: */
    while (1) pause();
}
```

# In conclusion

By saving the state of all registers on the stack, the context switch mechanism can suspend a process so that it can be resumed later.

There's always some machine-dependent intricacy to this, but the outline is usually the same.

We have separated the *mechanism* of context switch from the *policy* decisions about what process should run when.

# Scheduling and Messages

Mike Spivey

Hilary Term 2022


Department of
COMPUTER
SCIENCE

# Inside the kernel

For simplicity, the kernel of the operating system cannot be interrupted.

- So we can deal with one interrupt or system call at a time.
- I'll describe the internal data structures as a guide to what the kernel does.

# Process states

Each process is one of

- `ACTIVE` – running or ready to run.
- `SENDING`, `RECEIVING` – waiting to exchange a message.
- `IDLING` – the idle process.
- `DEAD` – after exiting.

Each process can be on *at most one* queue.

# The process table

```
struct _proc {
    int pid;              // Process ID
    char name[16];        // Name for debugging
    unsigned state;       // SENDING, RECEIVING, etc.
    unsigned *sp;         // Saved stack pointer
    int priority;         // Priority: 0 is highest

    proc waiting;         // Processes waiting to send
    int pending;          // Whether interrupt pending
    int msgtype;          // Message to send or receive
    message *msgbuf;      // Pointer to message buffer

    proc next;            // Next process in queue
};
```

# Process priorities

0: Device drivers.

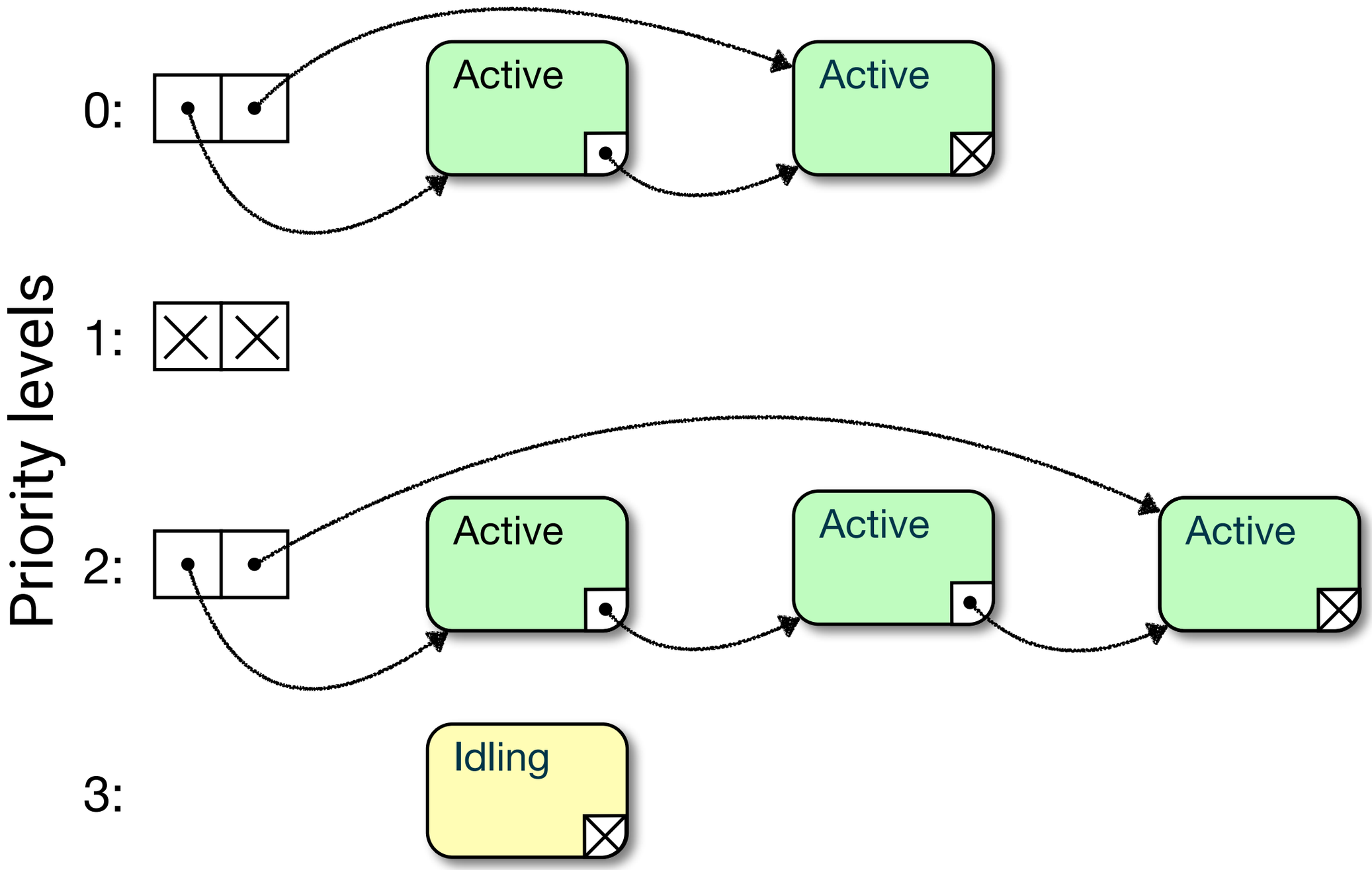
1: Normal processes, high priority.

2: Normal processes, default priority.

3: The idle process.

- When should a normal process be given high priority?

# Implementation: ready queues



# Pre-emptive scheduling

micro:bian is *pre-emptive*: a process can be suspended involuntarily (for example on interrupt), or when it calls `send()` or `receive()`.

Scheduling is not time-based: there is no attempt to share time equally between ready processes.

A process can call `yield()` voluntarily, but this is rarely necessary.

You can run a timer task if you like, but it is not compulsory.

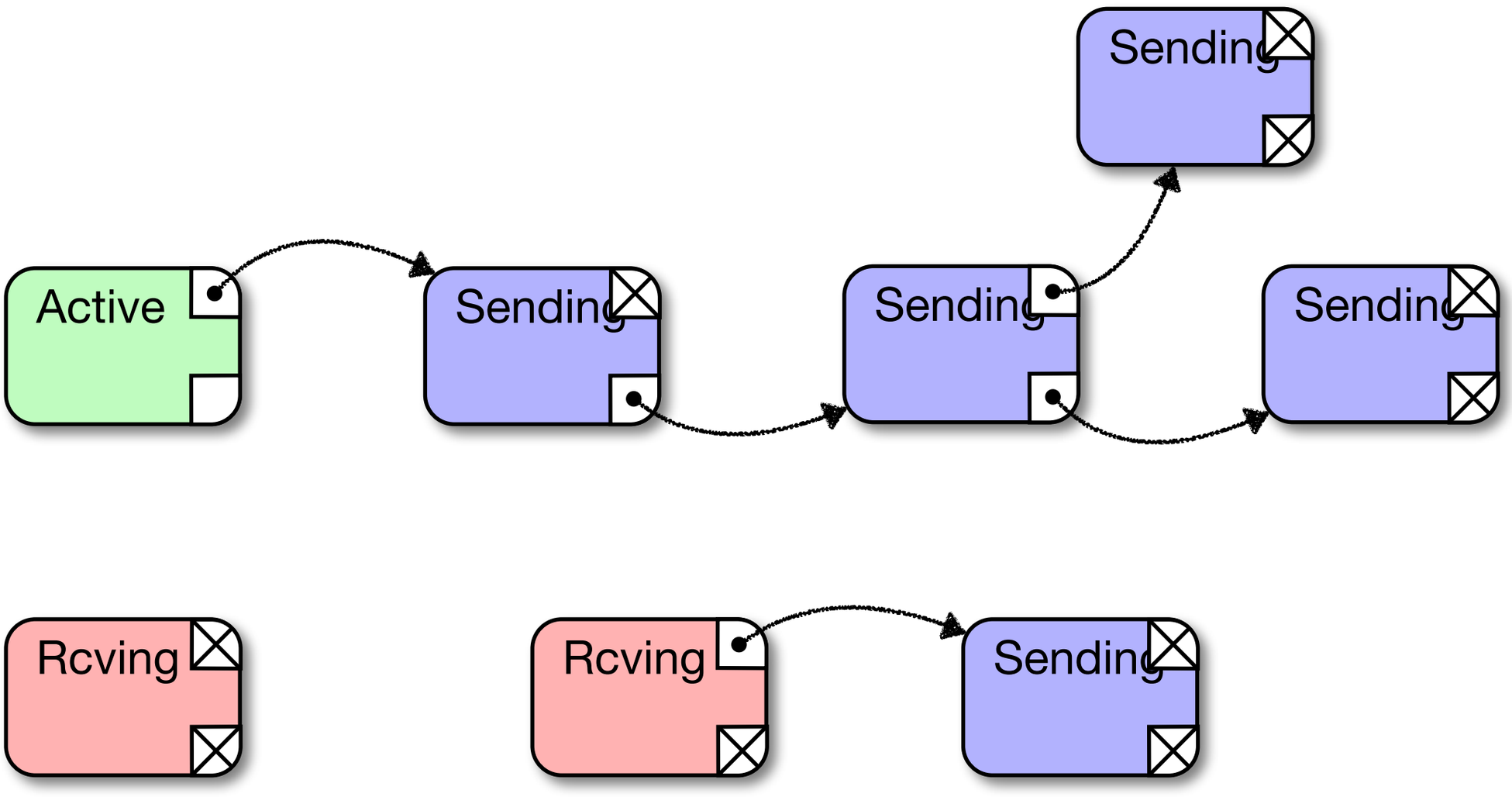
# Rendezvous principle

Two processes must both arrive at `send()` and `receive()` for a message to be passed.

- So some processes are waiting to receive – not on any queue.
- Others are waiting to send to a specific receiver – on a queue for that receiver.

# Sending queues

Each process has a queue of others waiting to send to it.



# Story of a system call

```
unsigned *system_call(unsigned *psp) {
    short *pc = (short *) psp[PC_SAVE];
    int op = pc[-1] & 0xff;

    os_current->sp = psp;

    switch (op) {
    case SYS_SEND:
        mini_send(arg(0, int), arg(1, int),
                  arg(2, message *));
        break;
        ...
    }

    return os_current->sp;
}
```

# Implementing send

```
static void mini_send(int dest, int type,
                      message *msg) {
    int src = os_current->pid;
    proc pdest = os_ptable[dest];

    if (accept(pdest, type)) { // Receiver is waiting
        deliver(pdest->msgbuf, src, msg);
        make_ready(pdest); make_ready(os_current);
    } else { // Sender joins the receiver's queue
        set_state(os_current, SENDING, type, msg);
        enqueue(pdest);
    }

    choose_proc();
}
```

# Choosing the next process

```
static inline void choose_proc(void) {
    for (int p = 0; p < NPRIO; p++) {
        queue q = &os_readyq[p];

        if (q->head != NULL) {
            os_current = q->head;
            q->head = os_current->next;
            return;
        }
    }

    os_current = idle_proc;
}
```

# Implementing `receive`

```
static void mini_receive(int type, message *msg) {
    if (os_current->pending     // Is an interrupt due?
            && (type == ANY || type == INTERRUPT)) {
        os_current->pending = 0;
        deliver(msg, HARDWARE, INTERRUPT, NULL);
    } else {
        proc psrc = find_sender(os_current, type);
        if (psrc != NULL) {     // Is a sender waiting?
            deliver(msg, psrc->pid, psrc->msgbuf);
            make_ready(os_current); make_ready(psrc);
        } else {                // No luck: we must wait
            set_state(os_current, RECEIVING, type, msg);
        }
        choose_proc();
    }
}
```

# Joining the queue

```
static inline void enqueue(proc pdest) {
    os_current->next = NULL;
    if (pdest->waiting == NULL)
        pdest->waiting = os_current;
    else {
        proc r = pdest->waiting;
        while (r->next != NULL)
            r = r->next;
        r->next = os_current;
    }
}
```

# Implementing a device driver

Mike Spivey

Hilary Term 2022


Department of
COMPUTER
SCIENCE

# A temperature sensor

The Nordic chip has a temperature sensor on the processor die.  We will implement

```
int temp_reading(void)
```

Use a device driver process

- to manage concurrent access,
- to allow connecting to interrupts.

# Hardware registers

There's a device TEMP (address 0x4000C000) with

- A *task* START (offset 0x000) to start a reading.
- An *event* DATARDY (offset 0x100) that signals the reading is ready.
- A register INTEN (offset 0x300) where we can enable an interrupt on DATARDY.
- A register TEMP (offset 0x508) where the reading appears.

# Describing the hardware

In `hardware.h`:

```
/* Temperature sensor */
#define TEMP_START          _REG(unsigned, 0x4000c000)
#define TEMP_DATARDY        _REG(unsigned, 0x4000c100)
#define TEMP_INTEN          _REG(unsigned, 0x4000c300)
#define TEMP_VALUE          _REG(unsigned, 0x4000c508)

#define TEMP_INT_DATARDY 0
```

# In assembly language

An assignment like

```
temp = TEMP_VALUE
```

becomes

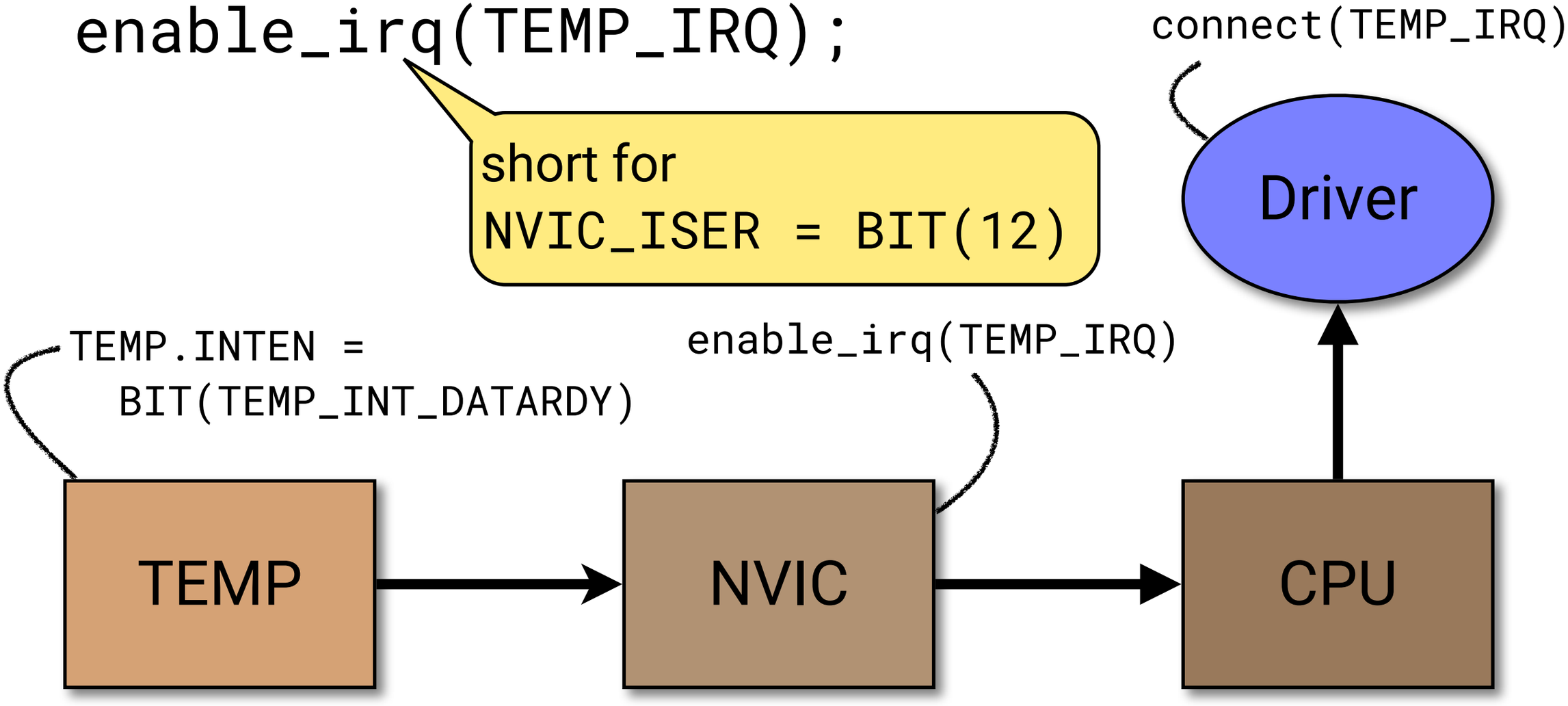
```
ldr r0, =0x4000c508
ldr r6, [r0]
```

and that’s all that matters.

# Configuring the interrupt

```
TEMP.INTEN = BIT(TEMP_INT_DATARDY);
connect(TEMP_IRQ);
enable_irq(TEMP_IRQ);
```

short for
`NVIC_ISER = BIT(12)`



# The driver process

```
static void temp_task(int arg) {
    message m;
    int temp, client;

    TEMP_INTEN = BIT(TEMP_INT_DATARDY);
    connect(TEMP_IRQ);
    enable_irq(TEMP_IRQ);

    ... server loop ...
}
```

# The server loop

```
while (1) {
    receive(ANY, &m);
    switch (m.type) {
    case REQUEST:
        client = m.sender;

        ... take a reading ...

        m.int1 = temp;
        send(client, REPLY, &m);
        break;

    default:
        badmesg(m.type);
    }
}
```

# Taking a reading

```
TEMP_START = 1;
receive(INTERRUPT, NULL);
assert(TEMP_DATARDY);
temp = TEMP_VALUE;

TEMP_DATARDY = 0;
clear_pending(TEMP_IRQ);
enable_irq(TEMP_IRQ);
```

- one request at a time, so we can stop to wait for the interrupt.

# A client subroutine

```
int temp_reading(void) {
    message m;
    sendrec(TEMP_TASK, REQUEST, &m);
    return m.int1;
}
```

short for

```
send(TEMP_TASK, REQUEST, &m);
receive(REPLY, &m);
```

# Why `sendrec()`?

- It’s a handy abbreviation.
- It is slightly more efficient – avoids two context switches.
- It solves the problem of *priority inversion*.

# Priority inversion

- The server receives a request, takes a reading, then tries to send the result to a client.
- Meanwhile, the client has been squeezed out by other processes, and has not run again: it has yet to reach its `receive(REPLY, ...)`.
- So the (high-priority) server must wait for the (low-priority) client before continuing.

Solution: use `sendrec()` so the client promises in advance to accept the reply.

# Links

- *Concurrent Programming* – programs that exploit parallel processing.
- *Concurrency* – proving that concurrent programs e.g. don't suffer from deadlock.
- *Compilers* – automating the translation into assembly language.
- *Computer Architecture* – designing hardware for performance.

# Next term

- First half: designing combinational and sequential logic circuits.
- Building a tool-kit of architectural elements.
- Second half: designing a datapath and control that can execute (some) Thumb instructions.